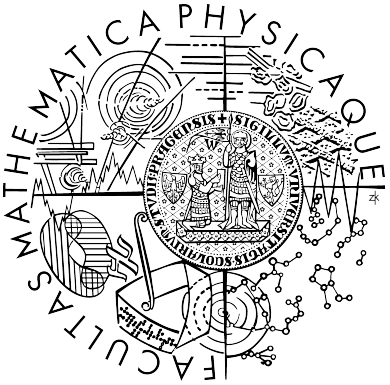
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Jiří Täuber

Deployment of Performance Evaluation Tools in an Industrial Use Case

Department of Distributed and Dependable Systems

Supervisor: doc. Ing. Petr Tůma, Dr.

Study programme: Computer Science
Specialization: Software Systems

Prague 2012

First of all, I would like to thank my advisor for his patience and valuable suggestions, all the people who revised this work, my family for their support in my life and studies, and also the people and who helped me with BEEN tryouts in both Seznam.cz and Koukaam a.s.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague on December 6th 2012 Jiří Täuber

Název práce: Nasazení nástrojů pro měření výkonu software do praxe

Autor: Jiří Täuber

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: doc. Ing. Petr Tůma, Dr.

Abstrakt: Výkonnost software dnes měří nejen specializované firmy v rámci recenzí, ale čím dál častěji je to běžnou praxí i pro samotné vývojáře aplikací. Firmy jsou často nuceny vyvíjet a udržovat vlastní nástroje pro měření vyvíjených aplikací. Na Matematicko-fyzikální fakultě vznikl nástroj pro automatizaci měření software jménem BEEN, který by měl správu jednotlivých měření významně usnadnit, ale jeho skutečný význam není možné vyzkoušet v prostředí, kde vznikl.

Cílem této práce je nasadit BEEN do reálného prostředí komerční firmy a vyhodnotit použitelnost tohoto nástroje pro vývojáře. Zaměříme se na vyhodnocení objektivních i subjektivních kladů a záporů, které používání tohoto nástroje mělo pro nezaujaté uživatele.

Klíčová slova: BEEN, benchmark, měření výkonu, praxe

Title: Deployment of Performance Evaluation Tools in an Industrial Use Case

Author: Jiří Täuber

Department: Department of Distributed and Dependable Systems

Supervisor: doc. Ing. Petr Tůma, Dr.

Abstract: Nowadays software performance is evaluated not only by specialized review companies but it is more and more starting to be a common practice for the software developers themselves. Companies are often forced to develop and maintain their own tools for measuring performance of the developed applications. On the Faculty of Mathematics and Physics there has been created a toolkit for automation of software performance evaluation called BEEN. This toolkit should significantly ease the management of individual performance measurements but it is not possible to test it thoroughly in the environment where it was created.

The goal of this thesis is to deploy BEEN in a real environment of commercially oriented company and evaluate the usability of this toolkit for the developers. We will focus on evaluating both objective and subjective positives and drawbacks of this toolkit as observed by unbiased users.

Keywords: BEEN, benchmarking, performance measurement, deployment

Table of Contents

Chapter 1: Introduction

1.1. Performance evaluation

Software performance measurements are starting to be a common practice for software developers. Companies are often forced to develop and maintain their own tools for measuring performance of the developed applications. We are going to test a toolkit for automation of software performance evaluation called BEEN. This toolkit should significantly ease the management of individual performance measurements.

Performance evaluation relies on several statistical theories. Detailed description of these theories is beyond the scope of this thesis. We will define only the basic terms and principles to understand the nature of our use case.

1.1.1. Benchmarking

*"In computing, a **benchmark** is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it. The term 'benchmark' is also mostly utilized for the purposes of elaborately-designed benchmarking programs themselves."* [1]

*"A **software regression** is a software bug which makes a feature stop functioning as intended after a certain event (for example, a system upgrade, system patching or a change to daylight saving time). A **software performance regression** is a situation where the software still functions correctly, but performs slowly or uses more memory when compared to previous versions."* [2]

Regression benchmarking is the process of repeatedly running a standardized performance test in order to uncover performance regressions in a computer system. In this thesis we will deal only with the performance of software applications or their parts.

1.1.2. Versions, Builds, Runs

In regression benchmarking we need to measure the software at different stages of the development process and observe how the measured parameters changed. A natural way of doing this is by repeating the measurement every time there is a significant change in the code. We expect the measured software to be maintained by some revision control system which marks changes in the program as different ***revisions*** also called ***versions***.

Modern processors contain multiple levels of cache memory. In processor and memory demanding applications, the performance is affected by the structure of data that are being evaluated. The structure of the data is determined by the compiler. Therefore for high precision measurements, each version needs to be compiled several times in order to negate random effects introduced by compilers as described in [3]. The process of each compilation is called ***build*** and the result is a ***binary***.

A robust evaluation requires collecting a representative set of samples. This is achieved by running the same binary multiple times. This should help average over

the random effects caused by the operating system, hardware interrupts and other aspects outside of the measured application.

1.1.3. Benchmarking In Production Environment

Measuring a software during development or after a release is a common practice nowadays. There are many projects that compare specific software types. For example there are simple benchmarks for compression speed and ratio of different archival tools (e.g. PeaZip), complex benchmarks for measuring various aspects of any CORBA® implementation (Xampler), or workload generators for a specific application (e.g. Rubis benchmark).

Most software companies use their own performance evaluation tools. These tools are specialized pieces of software. They are capable of evaluating one specific product or even only one aspect of it. Not very often we can see any generalized benchmarking software used in a company. This approach means there is a lot of work needed for measuring each application. Each benchmarking application needs to be written from the scratch. The large work overhead required by each measurement leads companies to perform only basic measurements.

1.2. BEEN

*"In computer programming, a **software framework** is an abstraction in which software providing generic functionality can be selectively changed by user code, thus providing application specific software. A software framework is a universal, reusable software platform used to develop applications, products and solutions. Software Frameworks include support programs, compilers, code libraries, an application programming interface (API) and tool sets that bring together all the different components to enable development of a project or solution."* [4]

BEEN is a benchmarking framework developed by teachers and students of the Faculty of Mathematics and Physics that tries to serve as a foundation for easy development of new benchmarks. The main goal of this framework is to automate regression benchmarking and provide support for common tasks in the benchmarking process. It contains some generic tasks in complete packages and eases the development of new packages by providing code base and powerful execution environment. This framework should simplify especially the most complicated and distributed measurements. We are aware that BEEN may not be the best solution for companies that already have their own benchmarking tools but we believe that BEEN may significantly reduce the programming time required for developing benchmarks in companies that are developing their own benchmarking toolkit from scratch.

1.2.1. Overview

BEEN is a framework that expects benchmarks to have a the structure explained in the chapter 1.1.2. Each measurement contains multiple versions, each version is built multiple times and each binary is ran multiple times to get the most precise measurements possible. BEEN's core services provide basic functionality required by all types of benchmarks that have this structure (running tasks, storing and retrieving measured data, etc.). These services also create the infrastructure for easy configuration and debugging of BEEN and any external packages ran by it.

The downside is that BEEN is a toolkit that does not measure any particular performance indicator by itself. It has the ability to run the Xampler benchmark included in the distribution which serves as an example. Its main potential is in easy extensibility and configurability.

BEEN is a distributed framework written in Java so it can be executed almost anywhere where Java Runtime Environment is present. There are exceptions to this general rule but they can usually be solved by changing the configuration of the underlying system.

1.2.2. Tasks and Pluggable Modules

In BEEN every action is carried out by a small application-like package called ***task***. BEEN tasks can be either one-time ***jobs*** or long-running ***services***. They can be written in Java, Python or a shell script (any script executable directly by the underlying operating system). These tasks are packed in `*.bpk` files and executed by the BEEN core (namely the Host Runtime).

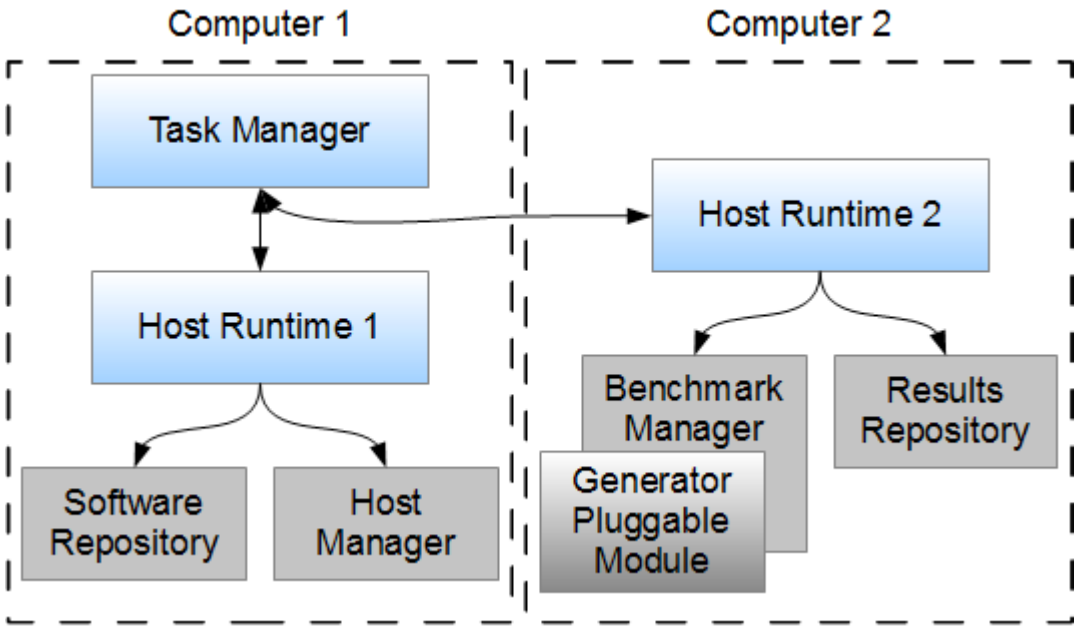


Figure 3.1: Schematic representation of the relation between core BEEN components.

Tasks tend to share some code so BEEN offers the option to write ***pluggable modules*** which are equivalents of dynamically linked libraries. Pluggable modules are also packed in `*.bpk` files and differ from tasks only by their internal structure and included meta data.

1.2.3. Host Runtimes and Task Manager

The core of BEEN consists of one or more ***Host Runtimes***. Each *task* is executed by a *Host Runtime* as a separate process. This may be either a new Java Virtual Machine for Java and Jython *tasks* or it may be plain execution of a shell script *task*. As the name suggests there may be more *Host Runtimes* running on different host machines at the same time for easy distributed computing. Running more *Host Runtimes* is useful either for client – server architecture measurements or for plain distribution of responsibility on different hosts.

Host Runtimes need to communicate and coordinate their actions. For this purpose we have a central *service* in BEEN called ***Task Manager***. *Task Manager* is the only BEEN *service* that is not executed by any *Host Runtime*. *Task Manager* is a standalone application that has to be executed even before any *Host Runtime*. Each *Host Runtime* needs to connect to the *Task Manager* when it starts.

Task Manager schedules and delegates the work on the connected *Host Runtimes*. It decides which *task* should be executed, when it should be executed and which host should run it. It also keeps track of running and finished *tasks*, stores their logs and provides additional functions for *task* management.

1.2.4. Core Services

Task Manager and *Host Runtimes* are the core of BEEN framework but there are additional *services* required for running BEEN. These *services* are different from the *Task Manager* because they are already BEEN *tasks* executed by a *Host Runtime*.

Host Manager is a *service* that manages detailed information about hosts. It automatically schedules ***detector tasks*** on hosts to gather detailed information about host's hardware and operating system. It also helps the *Task Manager* to decide which host should be used for running a *task*.

Software Repository is a *service* that provides *Host Runtimes* with *tasks'* packages. *Host Runtime* distribution contains only the core *tasks* so they can be ran even without the *Software Repository* but all the additional *tasks* and *pluggable modules* that actually perform the benchmarking are kept only on one host and are downloaded on demand.

Results Repository is a central storage for all the results and their meta data. It is not a relation database even though its structure reminds it. It operates with ***datasets*** (table-like structures), ***data handle tuples*** (table-rows) and ***data handles*** of various types. Additionally the *Results Repository* is equipped with a storage for large files. *Tasks* are free to store and load their results data there. Some *datasets* may contain ***triggers***. These *triggers* are simple structures which contain only a condition and a ***task descriptor***. When a new data that satisfy the condition are saved into the *dataset* then the *task* described by the *task descriptor* is scheduled.

Benchmark Manager is a *service* that can manage multiple benchmarks at the same time. It will allow user to create, configure and schedule measurement experiments. It is especially useful for automatically running ongoing regression benchmarks.

1.3. Academic projects

We think that BEEN is a good and useful toolkit but we are also aware that applications developed on the academic grounds tend to have only a weak relation to the real world. Problems with usability in the real world originate in the small community of people who participate in the development and evaluation of the software. The main purpose of the assignments rarely is producing quality software but rather evaluating the programmers skill. Moreover, people who are involved in the development process often idealize the application they make. They also subconsciously overlook or avoid bugs that they know of. These bugs may have easy workarounds but they can still pose a significant obstacles to an inexperienced user.

In general there are more reasons why the academic projects are less usable in the production environment. They include limited number of testing platforms and use cases for testing, not enough emphasis on quality and stability of the result, sacrificing simplicity in favor of experimental technologies, etc.

1.4. Premise of this thesis

To see how well has BEEN really been created as a widely usable application, we have to test it outside the department that created it. Unfortunately the scope of this thesis does not allow for extensive testing in various production environments. To get the most out of limited amount of tryout samples we need an environment that is significantly different from the academic grounds.

Industrial environment is inherently oriented at different goals than academic. Only a few simple requirements on the company should give us reasonably good testing environment to determine whether BEEN is usable by the general public or not.

Chapter 2: Project Definition

2.1. The Goal

The goal of this thesis is to deploy the current version of the BEEN performance evaluation toolkit in an industrial environment, with the intent of evaluating practical applicability of the toolkit. It is expected that some changes and additions will be necessary for successful deployment of the application, however evaluating the deployment demands and benefits for the company will be the main result of this thesis.

User experience would be best evaluated by psychometric or psychological questionnaires. Unfortunately both of the mentioned methods rely on statistics which require at least 5 people participating in the research for the results to have any weight at all. Creating a psychometric questionnaire is a work for several people and takes about 6 months [5]. Other psychological questionnaires may be less demanding but they still require work of professionals which is not available for this thesis.

Rather than an exact numerical evaluation of BEEN's viability, we want to know specific areas that we can improve on. We will use an user experience evaluation common in production software that is gathering bug reports and user feedback by asking users about their experience with this framework.

2.2. Deployment Requirements

We have already established that industrial environment is going to be the base for the deployment but we still need to determine some additional requirements before selecting the company. Some of the demands are simple and obvious but stating them will help us make a clear decision.

2.2.1. Requirements for the company

RC-1. It has to be an IT industry
Ideally it should be an IT developer company with their own applications to test. It is possible to test third party applications but that would lead us one step closer to the Academic environment where we are measuring mostly third party applications.

RC-2. It should not be a small company
One of the dominant aspects of the Academic environment is the small community. By introducing BEEN to a larger company we will try to test it in a more structured work environment where several independent developer groups interact to accomplish larger goals.

RC-3. The company must be willing to try BEEN out
As trivial as it seems this is probably the most limiting factor in the company selection. The deployment will require cooperation from the company employees therefore this tryout is not absolutely cost-free for the

company. To reduce the impacts of this requirement we will try to minimize the work required from the company developers.

2.2.2. Requirements for the tested application

RA-1. It has to be easily measurable
BEEN is a universal benchmarking toolkit but some applications are inherently hard to measure. In our trial will be able to measure only the most basic parameters. They may be response time, CPU utilization or analysis of data that the application saves in its log.

RA-2. The measurement has to be complex enough
The goal is to show and evaluate the strength of a complex toolkit. It is only natural that it should be used on complicated benchmarks. In fact we do not preclude the use of simple single-use benchmarks programmed from scratch. We just believe that BEEN greatly reduces the number of benchmarks that are worth writing as single-purpose. This demand goes against the previous one so we need to find a compromise between the two.

RA-3. The application should be distributed
This demand is tied to the previous one. BEEN has been developed to support distributed benchmarking. Using it on single-machine benchmarks is possible but it would show only a part of the toolkit's benefits. This is only a matter of preference therefore the final evaluation of our deployment evaluation will not be affected by this demand.

RA-4. There should be more than one working version of the application
To fully demonstrate the power of BEEN to our pilot users we would like to use the regression measurements capability of the toolkit. This is only possible if there are more revisions of the software to measure.

2.3. Deployment Scenario

Let us introduce the perfect scenario for BEEN deployment. Of course in real situation we will be limited by the company and their resources. We are aware of the limitations from the start and we discuss them in this section to provide the same knowledge we had at each state of the deployment process.

2.3.1. The Ideal Case

In an ideal situation we work with company that is highly interested in introducing BEEN into their regular development process. After the initial presentation of the toolkit this company dedicates a few developers to get familiar with BEEN. They use only the documentation and online resources to learn BEEN without further assistance.

After getting acquainted with BEEN, the developers decide to start regression benchmark either on a middleware[1] implementation or on a server application. They will design and implement the necessary BEEN *tasks* and *pluggable modules* to carry out the measurements and data evaluation. After the initial experiments they will

[1] *"In a distributed computing system, middleware is defined as the software layer that lies between the operating system and the applications on each site of the system."* [6]

further tune the benchmark to get optimal results. When the benchmark produces some output, the company developers provide the benchmark results for the purpose of this thesis and they fill out a simple questionnaire. The questionnaire would be put together in cooperation with a psychologist to get reliable results.

2.3.2. Real Life Expectations

In the real situation we are limited by the company motivation, resource availability and the scope of this thesis.

We can expect the company to be indifferent to trying out the project. At best we can expect the company to have some sort of "*we will use it if it goes well*" kind of attitude. That means BEEN will not have any priority over the everyday work in the company. Consequently the developers probably will not be able to dedicate much of their time to BEEN which will significantly limit their ability to participate in the tryout. The limited participation on company developers' part implies that more analysis and implementation work will be required from me.

There are no funds tied to this thesis to pay a professional psychologist to create the output questionnaire. The questions will be put together with the best effort to get a sincere subjective evaluation of the toolkit and its usability.

The time allocated for the tryout will probably limit the ability to tune the benchmark. Therefore we can expect only illustrative measurement outputs. Producing only demonstration results will not have any impact on the usability evaluation.

Chapter 3: Deployment at Seznam.cz

3.1. Company Characteristics

Seznam.cz is the most visited web portal in the Czech Republic. It has over 600 employees [7]. The company develops its own server applications. The company was interested in evaluating BEEN as a replacement for their own specialized benchmarking framework which they had to bend and extend every time they wanted to perform new measurements. Their passive acceptance of this project was enough to carry out our experiment.

Let us evaluate Seznam.cz in respect to the requirements introduced earlier:

RC-1. Seznam.cz is an IT company. They do develop their own servers and even middleware implementations. They release some of their products as open source software.

RC-2. Having several hundred employees it is big enough. We can assume that a big enough portion of the employees are programmers and people directly involved in software development.

RC-3. The branch manager in Brno agreed to give the local programmers free hand in trying out BEEN as long as they perform their other work on schedule. This is one of the best results we could hope for in any profit-oriented company.

To fill in the context we should say that in Seznam.cz developers use C, C++, Python, PHP and Bash scripts as their main programming languages. BEEN supports Jython (Python ran in Java) and shell script tasks but the core is written in Java. This will give the developers a hard time orienting in the BEEN code.

3.2. Use Case Definition

In cooperation with the company developers we have defined the use case for trying out BEEN. We worked together because the company's developers know the tested software and the used environment, but they lack the detailed knowledge of performance evaluation theory and don’t know BEEN capabilities.

Company's developers have recently finished new version of hint server for their news server and they wanted to know how much load it can handle. That means how many requests per second it can serve when deployed on a particular server hardware. None of their performance measurement tools were able to test load of a fast server because they were not able to generate sufficient load from a single client and they were not able to coordinate multiple clients. The developers were happy to learn that BEEN is built for distributed measurements.

The deployment scenario involves the tested hint server and configurable number of clients. The server is written in C and communicates through FastRPC[1] protocol. Its job is to receive a request string, search its database and return a list of

[1] FastRPC is Seznam.cz implementation of remote procedure calls based on XMLRPC syntax. For more details see http://fastrpc.sourceforge.net/.

hint structures. The content of the hint server database may change in time but we take it as a constant parameter because when we perform any regression benchmarks then we will have the same database content for all the measurements. Clients use FastRPC-netcat application to connect to the server. For the whole process we will be able to use number of virtual machines with Linux[1] operating system as well as all the developer's stations whenever they are available. These hosts will be used for the server compilation, client execution and other support tasks. Aside from these virtual hosts which may have reduced computing power at times, we will have access to a dedicated physical server which will be used exclusively for the server deployment.

From the perspective of the BEEN framework it means the following tasks: First BEEN needs to download the hint server's source code from the company's SVN repository, compile it and build the installation package. This operation will be performed on one dedicated client host because it has to have several uncommon libraries installed in order to compile the server binary. After the server package has been compiled, BEEN will upload it to the *Results Repository* which completes the build part of the benchmark.

Next comes the running part of the measurement. BEEN downloads the server package to the dedicated server machine and installs the hint server there. BEEN also starts the server right after it is installed so it can start answering the requests. When the server is running, BEEN will start a pre-configured number of clients on the rest of available hosts which will generate the load on the server for a pre-configured period of time[2]. After all clients have stopped, the server will be shut down and BEEN will upload the server log to the *Results Repository* and uninstall the server.

Last part of the benchmark comprises of evaluating the measured data. BEEN will parse the server log and clients logs to determine how many requests were requested and how many the server has served. The data will be passed to R statistical software to compute statistics and create graphs.

3.3. Measurement Setup

3.3.1. Measured Property

We are measuring requests per second (RPS in short) that the measured server is able to serve. The served RPS is our dependent variable. As the independent variable we can chose either time (for unstable servers) or RPS received by the server.

For our experiment we expect the server to have stable performance in time so we will chose the received to served ratio. Unfortunately we have no reliable way of measuring the received requests on the server side. To solve this we will measure the number of requests sent to the server by a client. We know for a fact that we have a reliable intranet network with a negligible latency therefore all the lost requests will be the result of the server performance problems. The number of received requests is exactly the same as the number of sent requests.

[1] Debian distribution with selected packages
[2] See the chapter 3.3for details on how number of clients and time were determined

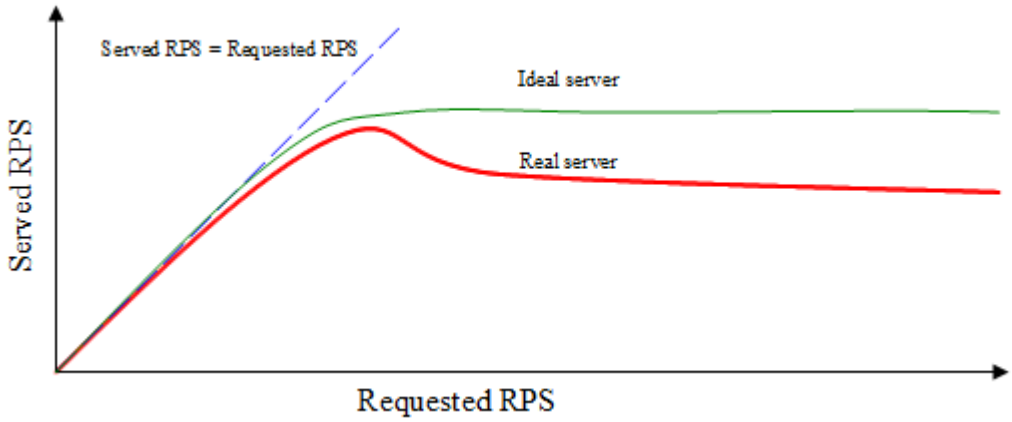


Figure 3.2: Expected graphical output of the request per second measurement. Blue dashed line is where the served RPS is equal to the received RPS. Note the drop in the real server‘s expected performance after the peak caused by server overload.

Note that the served RPS value will be roughly the same as the requested RPS value as long as the server is not fully saturated, but the served RPS will be lower once the server reaches its speed limits. Figure 3.2 shows the expected graphical output of the experiment.

3.3.2. Clients

We are measuring the dependence of served requests per second on requested RPS. Requests per second can be modeled by a ***poisson process*** [8]. The clients will send their requests in ***exponentially distributed*** intervals. To keep the intervals, clients will not be waiting for the response from the server. The properties of the exponential distribution allow us to easily divide the requests between many clients. Each client will have a set intensity of requests per second and the final intensity of requests is simple sum of the individual values.

3.3.3. Measuring Data

First we will need to determine the maximal speed of one client in order to prevent false results caused by clients sending less requests than expected. It is not a problem to add additional clients if we need to generate more load on the server because there is enough client machines connected by a very fast network.

Second step in our measurement process will be the rough assessment of the server’s maximal performance RPS. The goal is to measure the area where the server barely keeps up with the requests. The sent RPS values will be distributed around the notable bend in the figure 3.2.

To save the time required for measurements we will make the clients perform several measurements at different RPS values in one run. The basic premise is that the server’s performance is stable in time. Specifically for this case it means that measuring the response rate at different request rates does not have to be performed in different runs. We can have one long run with several intervals of different request rates and still get stable performance results.

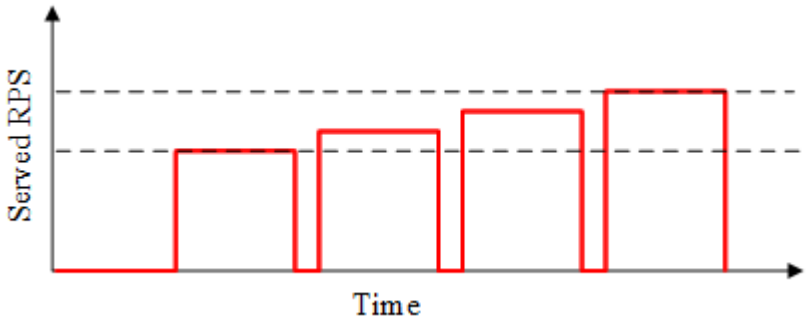


Figure 3.3: Visualization of client mean RPS changes in time. The RPS values are divided into equally long intervals separated by a brief pause. The pauses serve for client synchronization and for the measured server to process any requests from the previous burst before starting new measurement.

The clients will start at some configured minimal RPS load which will be determined in step 2, and will increase the load in steps until it is beyond the server's peak performance. There will be brief pauses between request bursts so the server can process all the requests from the previous burst before starting a new one. All the clients have to follow the same burst schedule for us to get useful results.

3.3.4. Processing Data

Information about the requested RPS comes directly from the clients. Clients log the time of each request which is easily parsed into RPS. We know that the times between individual requests are exponentially distributed because that's how we chose them.

Information about the served RPS will be determined from server's activity log which contains information about the incoming requests, their processing time, result and additional details that will not be used for our measurements. The server handles both found and not-found results with HTTP response[1] from 200 series. Timeouts and other failures are handled with HTTP responses 400 and higher. BEEN will therefore extract all the rows with request information from the log. Then it will filter the data and use only the entries from successfully answered requests.

From this basic data (RPS from client and successful RPS on server) we can plot the progress of each experiment. This plot can be used for reference when there are doubts about the experiment results. Data about the requested RPS and served RPS will be saved back in the BEEN *Results Repository* separately for each run and processed in the second step where we plot the final graph similar to figure 3.2.

3.4. Tuning BEEN

BEEN is still considered to be in the alpha stage of its development so there obviously were some bugs expected in the BEEN core. Fixing BEEN was not the main focus of this thesis but it was an issue we had to deal with during the deployment.

[1] HTTP response is a 3 digit number. 1st digit codes the general result, the last two digits code detailed result information. By standard response starting with the number 2 means success, response starting with 4 means an error on user part and response starting with 5 means an error on the server side.

The blocking bugs in the BEEN core took approximately 20 man-hours to locate and fix all together. Three additional critical bugs were discovered and put in the BEEN bug-tracking system. Two feature requests were made both from me and from the Company's developers.

3.5. Results Repository Datasets

Dataset design is similar to a database schema design in an application. For our simple experiment we tried to keep the *datasets* simple and very few. The *datasets* are explained in the following tables.

Tag name	DataHandle Type	Description
bin_file	`File`	UUID of the binary file stored in the *RR* File Store
version	`Int`	SVN revision number
build_time	`Long`	Unix timestamp of the time when the binary was compiled
run_count	`Int`	Number of times the binary has been executed

Table 3.1: The structure of the *dataset* ***binary***. This *dataset* is designed for storing the binary files and their meta data.

Tag name	DataHandle Type	Description
log_file	`File`	UUID of the server log file stored in the *RR* File Store
binary_id	`UUID`	Server binary UUID
rps_sequence	`String`	Sequence of the request intensities the clients were using
client_count	`Int`	Number of clients used for this measurement
server	`String`	Detailed server hardware information
database	`String`	Hint server database version or details

Table 3.2: The structure of the *dataset* ***server_log***. This *dataset* stores raw server log files, information about the server hardware and version of the hint database.

Tag name	DataHandle Type	Description
run_id	`Long`	Serial number of the run that generated this result
rps_requested	`Int`	Number of requests per second
rps_served	`Int`	Number of served requests per second

Table 3.3: The structure of the *dataset* ***parsed_rps***. This *dataset* stores preprocessed information about requested to served RPS ratio for each run.

Each of the *datasets* is intended to store results of one phase in the measurement process. The ***binary*** *dataset* stores compiled versions which are later downloaded, deployed and used for measurements. The ***server_log*** *dataset* stores logs of each measurement run. The logs are parsed and the information about served RPS is extracted second by second. Graph showing the progress of the measurement is created and stored in a web server directory in this step. The average of served RPS is computed for each burst from the same data and requested–served RPS pairs are stored back to the *dataset* ***parsed_rps***. All the data from the last *dataset* are used to plot the overall graph similar to the one in figure 3.2.

3.6. Implemented Tasks and Pluggable Modules

As we established in chapter 3.3, we will need to implement *tasks* that will build the measured hint server, deploy it, run several clients that will put enough strain on the measured server and in the end store the server logs in the *Results Repository*. The second part of our benchmark are the evaluation *tasks* which will need to gather the measured data from the logs and produce the output graphs.

There was never an attempt to use BEEN by someone else but the BEEN developers. All the *task* packages created for BEEN so far were incorporated directly in the BEEN core which is not a viable option for packages created by external developers. Each party creating packages for BEEN should be able to keep their own packages separated from others. This distinction was not possible in the original layout. Whole new directory for the externally developed packages has been created in the BEEN SVN repository. Files from this deployment scenario were placed in the new directory and can be used as a guideline for other cases.

When analyzing the problem and designing the BEEN *jobs* we tried to avoid using Java as much as possible and use shell scripting instead. The main reason was to allow the company's developers to participate in the development and maintain the code easily. However, it was necessary to implement *generator* and *evaluator pluggable modules* and some *jobs* in Java because coding them in Python or shell script would be too complicated.

3.6.1. Generator Pluggable Module

The first link in the process of creating the data is the *generator*. This *pluggable module* was written in Java. The *generator* creates the first two *datasets* (***binary*** and ***server_log***) when the analysis is created. On every experiment run, the *Benchmark Manager* runs this *generator* to determine the sequence of *tasks* that will be used to finish the experiment. The *task* sequence generated by our *generator* is visualized in the figure 3.4.

The *generator* contains very simple version of a planner which can be modified separately. The current planner simply takes a list of versions to test, compares it with already compiled and executed binaries and schedules new builds and runs of the least built version and the least ran binary. A more sophisticated planner could chose to re-evaluate versions which have statistically the least reliable results.

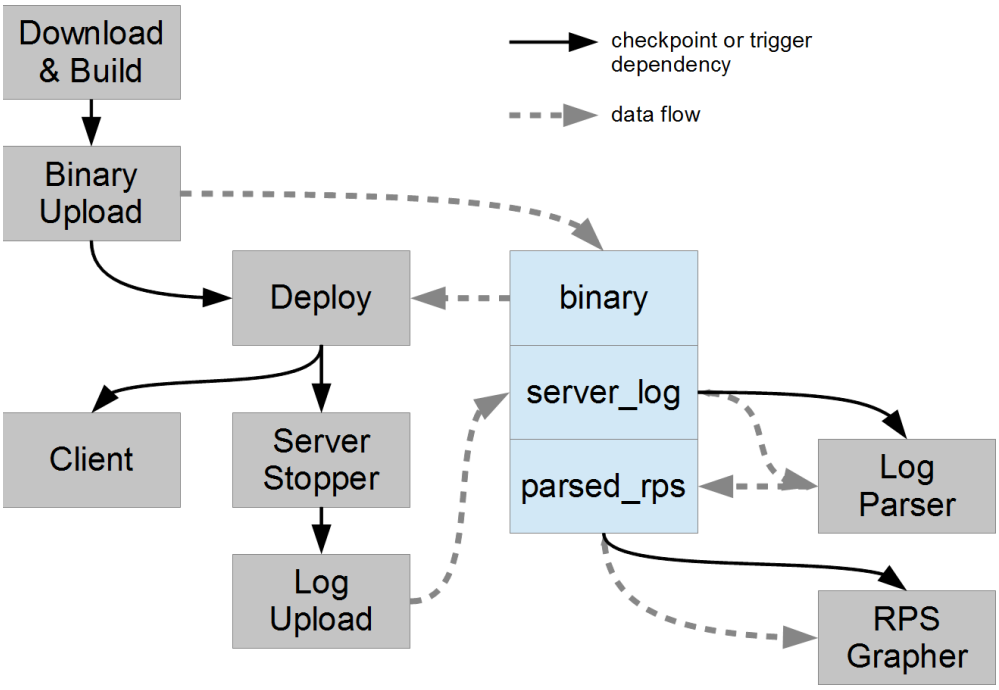


Figure 3.4: Flow chart of the *tasks* involved in the experiment. The blue boxes represent individual *datasets* in the *Results Repository* where the data are stored.

3.6.2. Download + Build Task

This *task* is a simple shell script *task*. It downloads the given version of the server from the SVN repository, builds it and stores the installation package on the local file system in a well-known location.

3.6.3. 3.6.3. Binary Upload Task

Shell script *tasks* are unable to access the *Results Repository* directly, therefore the previous *task* has to be coupled with a Java *task* which takes the compiled binary and saves it in the *Results Repository*. This *task* is set to run on the same host as the previous *task* to be able to locate the compiled server binary.

3.6.4. Deploy Task

Part of this *task* is written in Java and part is a shell script. First the Java code downloads the required binary to the sever machine. When the server binary package file is on the server, a shell script is executed to install and start the server.

3.6.5. Server Stopper Task

Right after the deploy task finishes, the stopper *task* is ran on the server. This shell script *task* waits for a given amount of time and then stops the server. It is passively waiting for the whole duration of the experiment. This *task* is set to be exclusive in BEEN because its main purpose is to prevent other BEEN *tasks* (including the detector *task*) to be scheduled on the server host while the experiment is running. Executing another *task* on the server machine could impact the server's performance.

3.6.6. Client Task

Writing clients in shell script would be complicated and Java is not necessary for this *task*. We chose Python for coding the clients as it provides good strength and is maintainable by the Seznam.cz developers. The client tasks call a XML-RPC client in a loop with specified intensity (exponential distribution of waiting times). This way they generate a predefined load on the measured server. There will be more than one client to fully saturate the server.

3.6.7. Log Upload Task

When the server stopper *task* finishes, the server log has to be uploaded to the *Results Repository*. Similar technique as with the binary upload is used. The two *tasks* differ only in the parameters they accept and in the *dataset* they save the file to.

3.6.8. Evaluator Pluggable Module

The previously mentioned log upload *task* is the last step in generating the measurement data. Evaluation of data starts with the *evaluator pluggable module* which creates the last *dataset* (***parsed_rps***), one *trigger* on the ***server_log*** *dataset* and one *trigger* on the ***parsed_rps*** *datatataset*.

3.6.9. Log Parser Task

This *task* is scheduled by the *trigger* placed by the *evaluator* on the ***server_log*** *dataset*. It has several parts. At first Java code downloads and parses the server log file. After the log file is processed and saved in a CSV format, the R script is launched to perform the statistical measurements and plot the measurement progress graph. The R script also produces the served RPS data and returns back to the Java code which takes the RPS data and saves them to the *Results Repository*.

3.6.10. RPS Grapher Task

The last *task* is scheduled by the *trigger* placed by the *evaluator* on the ***parsed_rps*** *dataset*. This *task* has a similar work flow to the previous *evaluator task*. At first a Java code downloads the parsed RPS data, then a R script is executed to process the data and plot the final graph.

3.7. BEEN Deployment

The company uses customized Debian Linux distribution for all their servers and development machines. For testing and debugging the BEEN *tasks* the company allocated two virtual servers. The biggest problem proved to be installing and configuring Java and the operating system so that BEEN can spawn additional JVMs and access the network properly.

3.7.1. JVM Tuning

In our first attempts the *Host Runtime* running on the main server was not able to start more than 4 *tasks* at any given time. The server had limited memory and an attempt to spawn more JVMs failed because the running *tasks* used nearly all of it. To

solve this we had to find a way to reduce the JVM memory requirements. At its start JVM allocates only a part of its heap memory and the whole stack memory. Limiting the heap memory would not help because it is raised automatically. Limiting the stack size seemed to be the better way.

By default the *Host Runtimes* themselves required at least 64 MB of memory allocated for the application stack. *Host Runtime* also started all the *task* JVMs with the same stack size requirement. We had to modify the execution script for *Host Runtimes* to set the JVM stack size for *Host Runtime* at its startup and we also had to make modifications to the *Host Runtime* itself so it would execute *tasks* with less memory than the *Host Runtime*.

3.7.2. Operating System Configuration

There were also problems with the reverse IP lookup and host name resolution. In the Java distribution on our system, the host name resolution used `localhost` entry to identify the host instead of any other interface. That caused communication problems between our *Host Runtimes* and the *Task Manager*. *Host Runtimes* send their own address and port to the *Task Manager* for callback purposes. The callbacks stop working when the address points to `localhost` instead of the external interface of the *Host Runtime*.

We did not find the reason why the reverse IP lookup did not work within Java. We found that clearing `/etc/hosts` file solved this problem but this solution is not recommended in Debian distributions.

Related to the previous problem was an issue with Java method `InetAddress.getCanonicalHostName()`. When it returned a symbolic host name, it returned it without a domain name. This forced us to use computers only from a single domain.

Once we found a solution to our problem, we did not look for the exact cause of the problem. Getting the framework to run was more important than looking for conceptual problems in its design. I tried to reproduce the problems later on but they did not occur in my testing environment.

3.8. Measurement

3.8.1. Task Debugging

Once the BEEN runtime environment was set up, we had to debug the measuring *tasks*. This proved to be more complicated than what the BEEN design promised. The *tasks* that created and gathered the data were mostly Bash scripts that we debugged outside of BEEN. For the final polishing in BEEN they were easy to start, restart and debug. Unfortunately we could run each *task* only once in each context which slowed the process.

On the other hand the triggered evaluating *tasks* were near to impossible to start manually. They require nontrivial parameters from the *Results Repository* to run properly and therefore they are very hard to debug. BEEN clearly misses the option to fire a *trigger* manually on demand.

3.8.2. Synchronization Problems

A big and unexpected problem was BEEN's inability to reliably start *tasks* as we would expect. It proved to be too complicated and unreliable to use BEEN's ***checkpoints*** to synchronize tasks. BEEN's synchronization mechanism lets one set a value to a *checkpoint* or wait for a specific value to be set to a *checkpoint*. We discovered that sometimes the *checkpoint* value was not sent to all the tasks waiting for it. More importantly we wanted to use a variable number of clients that needed to start simultaneously but only after the server was running.

Designing a reliable synchronization logic using *checkpoints* proved to be overly complicated. This concept is simply not suitable for multiple clients waiting for each other. We have discovered that even without any synchronization the clients started roughly at the same time right after the measured server was deployed. Instead of programming a complicated synchronization logic, we chose not to synchronize the *tasks* until we had a proof that our results are significantly affected by this. That allowed us to perform the measurements sooner but also added more complexity to the data evaluation process.

3.9. Measured Data

After two months of programming and debugging the above mentioned *tasks* we still did not have any valuable output. The branch boss therefore decided that adopting BEEN is no longer a viable option for the company and closed the tryout project. It means that we never had the chance to measure more than one version of the tested hint server and even that measurement was only in the stage of getting the initial data. The data mentioned in this section are the data we got from the real testing scenario while debugging the tasks.

3.9.1. Maximal Client Performance

The initial analysis required us to asses the maximal output of a single client. This was necessary for us to know how much load a single client can generate. We could have been surprised by the results later if we didn't do this measurement.

The first run of the client (viewed in figure 3.5) revealed that the system fork-bomb[1] protection is the limiting factor in the number of requests we would be able to send. It still gave us a reasonable first estimate of the client's potential.

[1] *"In computing, the **fork bomb** is a form of denial-of-service attack against a computer system which makes use of the fork operation (or equivalent functionality) whereby a running process can create another running process. [...] A fork bomb works by creating a large number of processes very quickly in order to saturate the available space in the list of processes kept by the computer's operating system."* [9]

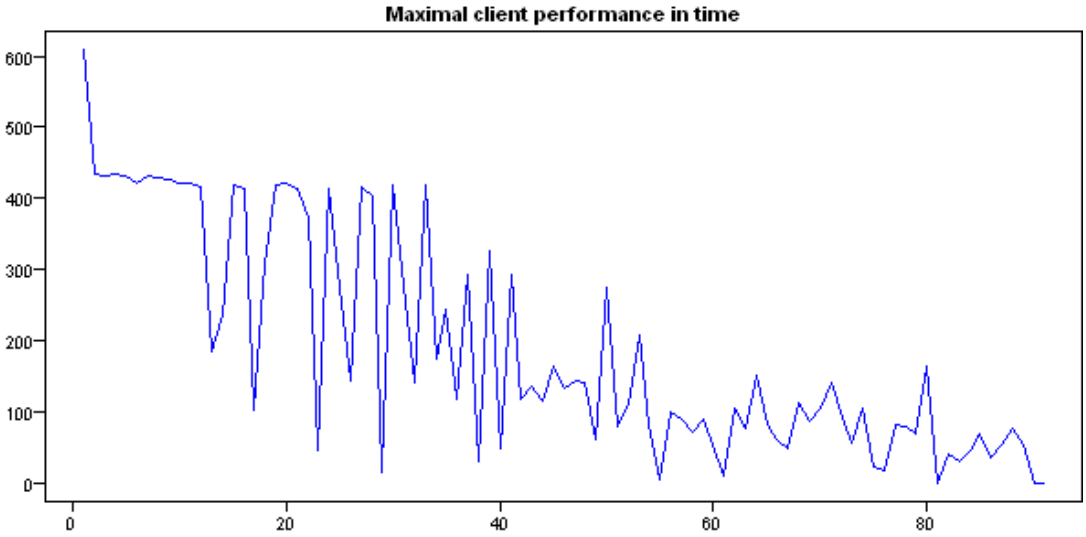


Figure 3.5: Visualization of requests generated by the client during a 90 seconds long period with no restrictions.

3.9.2. Client Performance in Time

We decided that we would not disable or alter the operating system protection just for this measurement. The administrative load that would be brought by this alteration was unnecessary and the result unsure at best. Instead of combating the operating system settings, we determined a good stable request rate for a client.

The same script that is ran by the BEEN framework was ran a few times by hand to determine a sustainable request rate. After a few tries we saw that 60 requests per second were sustainable for long enough time to complete our measurements. Figure 3.6 shows the results of our final observation.

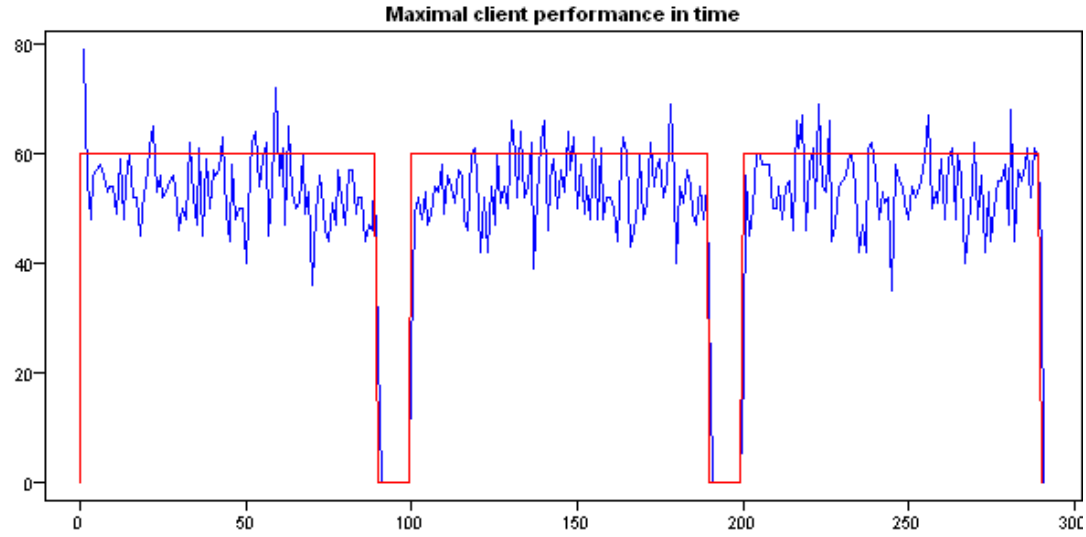


Figure 3.6: Visualization of requests generated by the manual run of client with 60 RPS as the parameter. Red line shows the expected request rate, blue line shows the log analysis.

With only 60 requests per second we would need too many clients to put enough load on the measured server. We were aware of this and we decided to postpone the optimizations until the whole benchmark is set up in a basic form.

3.9.3. Experiment Graph

The next stage of the benchmarking process required a test run of the whole benchmark. The result of the first *evaluator* run was generated partially by java code which parsed the raw server logs into CSV files. The CSV files were then processed by a R script. As the output we got three graphs displayed in figures 3.7 and 3.8.

For the testing run there were two clients used with only a short experiment duration. The measured hint server returned HTTP response code 200 for keywords it found in its database and HTTP response code 204 for the keywords that it did not find in its database.

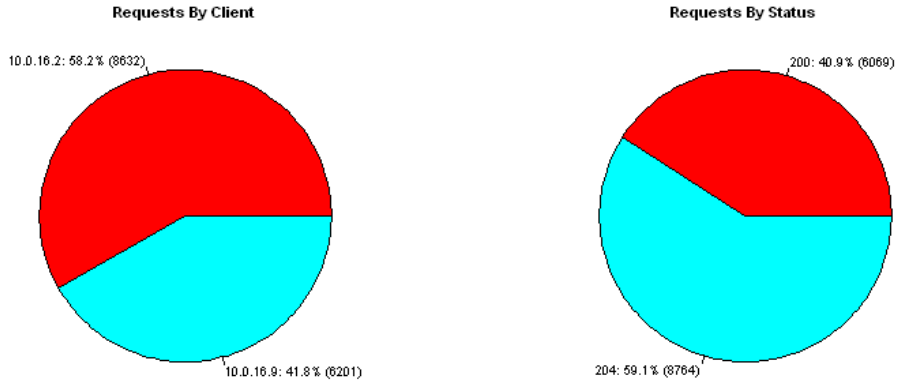


Figure 3.7: The pie charts that were produced as a result of a single experiment. The first graph shows the amounts of requests sent by each client. The second chart shows the percentage of server responses (database hits in red, database misses in blue)

These graphs illustrate that the algorithms provide valid results of our measurements. We can clearly see that one of the clients was faster in sending the requests despite the fact that both had the same parameters. We can also question our choice of the requested phrases. We would like the amount of positive and negative responses to be equal.

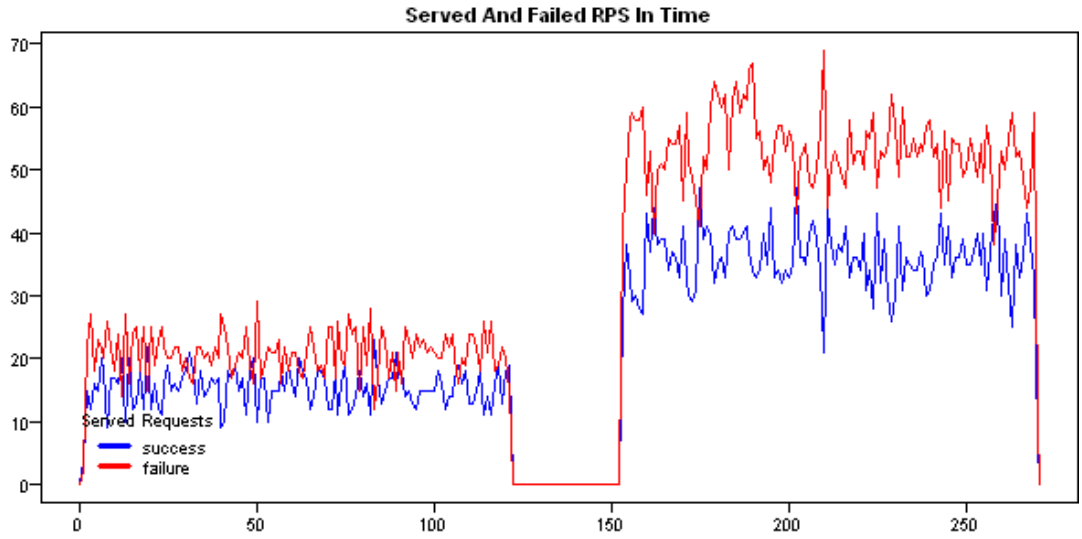


Figure 3.8: The time distribution of requests from the server's point of view.

As a result of this step we also gathered several files which were imported to the *Results Repository*. The data from these files are later used to generate the final graph of the benchmark.

3.9.4. Final Graph

The final graph is the ultimate result of the benchmark. It should have looked like figure 3.1. Unfortunately we were not able to debug the particular *evaluator job* that was responsible for processing the data.

Even using the existing code for generating the graph would not help. The graph would not have had any real information value because there were only a handful of experiments and too few data to generate this graph. The server was expected to be able to handle several thousand of requests per second. In the prototyping stage we had only handful of clients which could not generate enough requests.

3.10. User Experience

There is a big difference between the user who creates the benchmark *tasks* and the one who configures them. Most of the development work was done by me. Configuration of the benchmarks was done as a group effort to determine what was reasonable to configure and what would be better to embed in the code.

Our experiment involved only two people from the company which is not enough to get credible results by the scientifically correct methods mentioned in chapter 2.1. Therefore I used the SUMI questionnaire [5] as an inspiration and used common sense to evaluate the software based on the company employees' experience.

3.10.1. Developer's Perspective

The development process requires a lot of work and knowledge about the whole system. Even with the general knowledge of the system it is still hard to write the code efficiently. The total time spent on the development of this basic benchmark exceeded 200 man-hours which can hardly be considered adequate. I would expect that writing a benchmark in BEEN would take at most the same time as writing a proprietary benchmark. I would estimate that to be completed within 40 man-hours.

Writing the *tasks* as an external user who has no idea about the inner workings of the framework is possible but there are too few examples. The biggest development problem is probably the missing SDK[1] documentation. The general documentation document is probably the best source of information but it is not tied to the generated javadoc[2]. The javadoc is not complete and does not contain information about recommended practices. Overall there is no other option but to read the whole programmers guide and then start writing the code.

When the *tasks* start to emerge from the raw code, it is fairly easy to run both *services* and *jobs*. Debugging java *tasks* is unexpectedly easy thanks to the *Debug Assistant*. Scripted tasks are better debugged outside of BEEN before they are packed into the task package. Real problems start with *triggers* and *evaluator tasks*. The *evaluator tasks* expect to be started by a *trigger* and will not work properly when started manually. In BEEN there is currently no way to force the *Results Repository*

[1]Software Development Kit is typically a set of software development tools. Ussually an application programming interface, debugging aids, sample code and supporting technical notes.
[2] Java documentation generated from the source code comments. It can be generated from the BEEN source code package.

to activate a *trigger* which makes starting *evaluator tasks* almost impossible. Also when the *evaluator task* crashes during execution, there is no way of enabling the *trigger* again and the whole *Results Repository* has to be restarted.

We have used a lot of scripting *tasks*. Sometimes it was not very easy to connect the scripts to the rest of the BEEN framework. BEEN clearly lacks support for working with the *Results Repository* without writing java code. We have used the option to write the *task's* logic in a shell script. Then another java *task* was ran to upload the results into the *Results Repository*. When evaluating the result we used BEEN's ability to execute a shell script from java code. In that case we first downloaded the measured data from the *Results Repository*, ran a R script to evaluate them and then the java code uploaded the processed results back to the *Results Repository*.

The solution to the previously mentioned problem was supposed to be the BEEN *Command Line Interface*. Unfortunately the *CLI* client did not support IPv4 protocol. It had to be modified to allow for data export in an environment without the Ipv6 support. Also XML input/output that is used in the interface is generally good for archiving the data but it is not very friendly for shell scripting.

Overall the programming experience could be described as tedious but not very challenging. It needs a lot of BEEN knowledge and time before one is able to start programming useful *tasks*. Then there is a period of fairly mundane coding and debugging of the execution *tasks*. There are some conceptual issues that need to be solved during the development but they are not unique to BEEN. They are mostly related to programming for distributed computing environment. The programmer starts to encounter bigger problems only when debugging the *evaluator tasks*.

3.10.2. User‘s Perspective

A benchmarking software user is more interested in setting up the benchmark and viewing the results. In this sense the Seznam.cz developers were mostly BEEN users.

Users are mostly interested in the user interface which generally looked nice and well organized. Sometimes there were sections which contained more useful information than the user had expected. An example of such a section was the *Host Runtime* information page.

At first our users were a little confused with the *contexts* and tasks overview page. After understanding the principles, it became the center of all interest with the most important information about *task* progress. At the beginning we had a lot of *task* failures. The easiest way to determine the problem was to look at the *task* log accessible in this section. This part of BEEN's interface proved to be really well designed.

The benchmark overview section was not used as much as the *task* overview. Still it did not lack any vital information and was very useful for troubleshooting benchmarks. Problems with benchmarks not running on schedule were solved simply by looking at the list of running *contexts*. The individual benchmark setup pages were mostly designed by the benchmark developer. The benchmark setup interface was fairly easy to configure for the best user experience. We used default values in most of the fields for fast benchmark setup and we rarely changed them. It took less than a minute to create a new benchmark.

Results Repository section was the only one with some bigger user interface problems. The table details did not contain information about indexes. There is no information about *triggers* and most importantly the ability to fire a *trigger* manually is missing. Minor detail is that the data printout table is always shown on the page as a whole. Some paging structure may be more appropriate for large *datasets*.

Chapter 4: Deployment at Koukaam a.s.

After the early shutdown of our tryout in Seznam.cz I asked developers in Koukaam a.s. whether they wanted to try BEEN. Fortunately they were willing to try out the BEEN framework. This time we did not expect very complex measurements but that should give us faster results and prevent an early experiment shutdown as in the previous scenario.

4.1. Company Characteristics

KOUKAAM a.s. is the biggest distributor of IP camera systems in the Czech Republic since 2003. The company takes part in projects for both government and private sector. They have also introduced their own network video recorder (NVR in short) called IPCorder to the market. Another of their own devices is a network controllable power distribution unit, which allows to control virtually any electrical appliance via standard internet browser.

We can evaluate the requirements stated in the chapter 2.2:

RC-1. It is an IT industry. The company develops software for their devices. The network video recorder contains a server software which has high performance requirements.

RC-2. The company now has over 20 employees in their software development department divided into multiple teams. The division to multiple teams grants us the possibility to test in a structured environment.

RC-3. The need for a performance evaluation tool grew rapidly in the company so they were willing to try BEEN out as the first option. Most of the developers come from the Faculty of Math and Physics of the Charles University so they expected a tool with a familiar logic in the interface.

The programming languages used in Koukaam a.s. are C, C++, Java and JavaScript but for internal tools they prefer Bash or Python as the programming language. This is similar to Seznam.cz but unlike them, Koukaam a.s. developers will have easier time understanding the BEEN code because they also use Java in their day to day work.

4.2. Use Case Definition

During one of the tests the company testers realized that their embedded network video recorder sometimes creates gaps in the recorded video. After a series of tryouts, it was determined that the issue was caused by a performance problem. The device could not read the incoming data from the network, process it and save it to the hard drive fast enough. The recorded video contained gaps in the times where the device had temporary performance issues. The goal was to find and eliminate the performance issues so that the recorded video would be uninterrupted.

There are multiple variants of the NVR and all of them showed the same symptoms. The tested product portfolio ranges from KNR-090 which can record 4 cameras at once to KRR-424 which can record up to 24 cameras at once. As expected

the weaker platforms had bigger problems than the more powerful platforms. We decided to start measuring the weakest device but also think about the extensibility.

The deployment scenario involves only the tested NVR, a streaming server which will provide a stable and repeatable video stream and a management server for running BEEN. We will also use company's build server which already can automatically build specified firmware versions.

The benchmark scenario starts with a *task* that will communicate with the build server and will ask it to create a firmware of the specified version. This firmware will then be installed on the measured NVR. The NVR has a XML[1] API[2] which will be used to set up the device. We will connect it multiple times to the streaming server to simulate the required number of cameras. When the device is configured, it automatically starts recording. BEEN will then wait for some time for the device to collect sufficient amount of performance data. BEEN may send some XML requests and measure their response time as a responsiveness test while waiting for the experiment to finish. In the end the performance log will be loaded from the device into the *Results Repository* for further analysis.

Evaluation of the performance log will be the main work in this benchmarking scenario. We need to gather a lot of information that will be analyzed for possible performance issues. Choosing the right data to measure and evaluate will be essential.

As our output we want the experiment timeline which will show the changes of the measured properties in time. We will also want to see statistics of the measured properties: average, mean, the lowest, and the highest value.

4.3. Measurement Setup

4.3.1. Measured Properties

The device has multiple parameters that can be measured. After some initial tests we decided to measure the utilization of its CPU, RAM and the hard drives.

The first measured parameter was the ***CPU load***. It was measured using the standard Linux tools that read the property provided by the Linux kernel. CPU load is the main parameter in our measurement. It does not tell us directly how much the system is loaded. For example an average of 90% CPU load may not be an indication of a problem but only a proof of a well utilized hardware. More importantly 50% CPU load does not mean the device is performing 2 times better than with 100% load. On the other hand most of the failures may be explained simply by overloaded CPU.

We also measured the ***CPU I/O wait*** percentage. It gives the amount of time that the CPU was idle because all the running processes were waiting for some input/output operation. I/O operations include waiting for HDD to read or write something, waiting for network interface, memory, etc. We measure the disk load separately so this value helps us uncover unnecessary memory copies and any issues in networking.

---

[1] Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

[2] *"An **application programming interface (API)** is a protocol intended to be used as an interface by software components to communicate with each other."* [10]

The main function of a network video recorder is writing the video to a hard drive. How well the device does this is measured by the ***disk load***[1] property. Disk load and CPU I/O wait do have some correlation in a NVR but it is not absolute. Generally we would expect higher values of disk load than CPU I/O wait because there usually are processes that can run while the HDD is in use by another process.

The most complex measurement of a dependent unit are he frame drop sequences. We defined a ***frame drop*** as a gap in the recorded video stream. A frame drop is detected by a routine in the NVR when a delay between two received video frames is longer than it was supposed to be. We want to rule out possible network problems and temporary camera streaming problems, therefore we consider only gaps that are four times longer than expected. For example if we expect a video frame every 40 ms (25 frames per second) and we don't receive anything from a camera for 160 ms then we can relatively safely say that we missed some video frames. Shorter delays are mostly caused only by network delays or camera streaming issues. We measure the length of each frame drop sequence – the time between the two frames.

The above mentioned properties should tell us something about the reliability of the system. For a good user experience we also need the device to remain responsive. We can do that by measuring the time that the device needs to respond to a XML request sent usually by its web interface.

Along with these dependent variables we measured the total ***incomming traffic*** which is the measure of the total strain put on the device. This value is there mainly as a control measurement, which can reveal invalid or suspicious experiments.

4.3.2. Measurement Process

First we needed to generate sufficient load with reliable characteristics that would put enough repeatable strain on the measured NVR. A streaming server was implemented for this purpose.

The streaming server was used to simulate many real cameras at one time. Moreover it was able to provide parametrized video streams. We could chose the number of video frames per second, the video resolution, the encoding and most importantly the bit rate[2] of the video stream. We decided to cut the number of possibilities by fixing the number of frames to 25 per second and the resolution to 1920x1080 pixels. This left us with three variables: encoding, bit rate and the number of virtual cameras.

With the streaming server set, we can proceed to the firmware installation process. In our experiment, a special firmware version is installed in the NVR. This firmware is the same as the regular one, it only records the measured properties in the performance log every 10s. Each frame drop's length is recorded immediately in any version of the NVR firmware.

After the installation, the NVR is configured to connect to a predefined number of simulated cameras from the streaming server. The NVR records the video from all the virtual cameras. Typically it will be a number of cameras with the same

[1] Disk load is the amount of data that are being transferred to or out of the hard drive measured as a percentage of the maximal data transfer rate.
[2] The amount of data transmitted per fixed amount of time. The bit rate is quantified using bits per second (bps) unit.

bit rate and video encoding. After a predefined amount of time, the experiment is stopped, the log is extracted, stored in the *Results Repository* and analyzed.

4.3.3. Processing Data

The performance log that we get from the experiment contains one entry for each measured variable every 10 seconds. Measured variables are: CPU load, CPU I/O wait, disc load and incoming traffic for control purposes. On top of that, the same log contains information about all the frame drops.

We will use a python script to parse the performance log and gather the data required for visualizing the experiment progress. Unlike in Seznam.cz, this graph will be the main output of the measurement because it shows us the patterns in the device performance. The statistical data supplement the result with exact numbers which are hard to read directly from the line graphs.

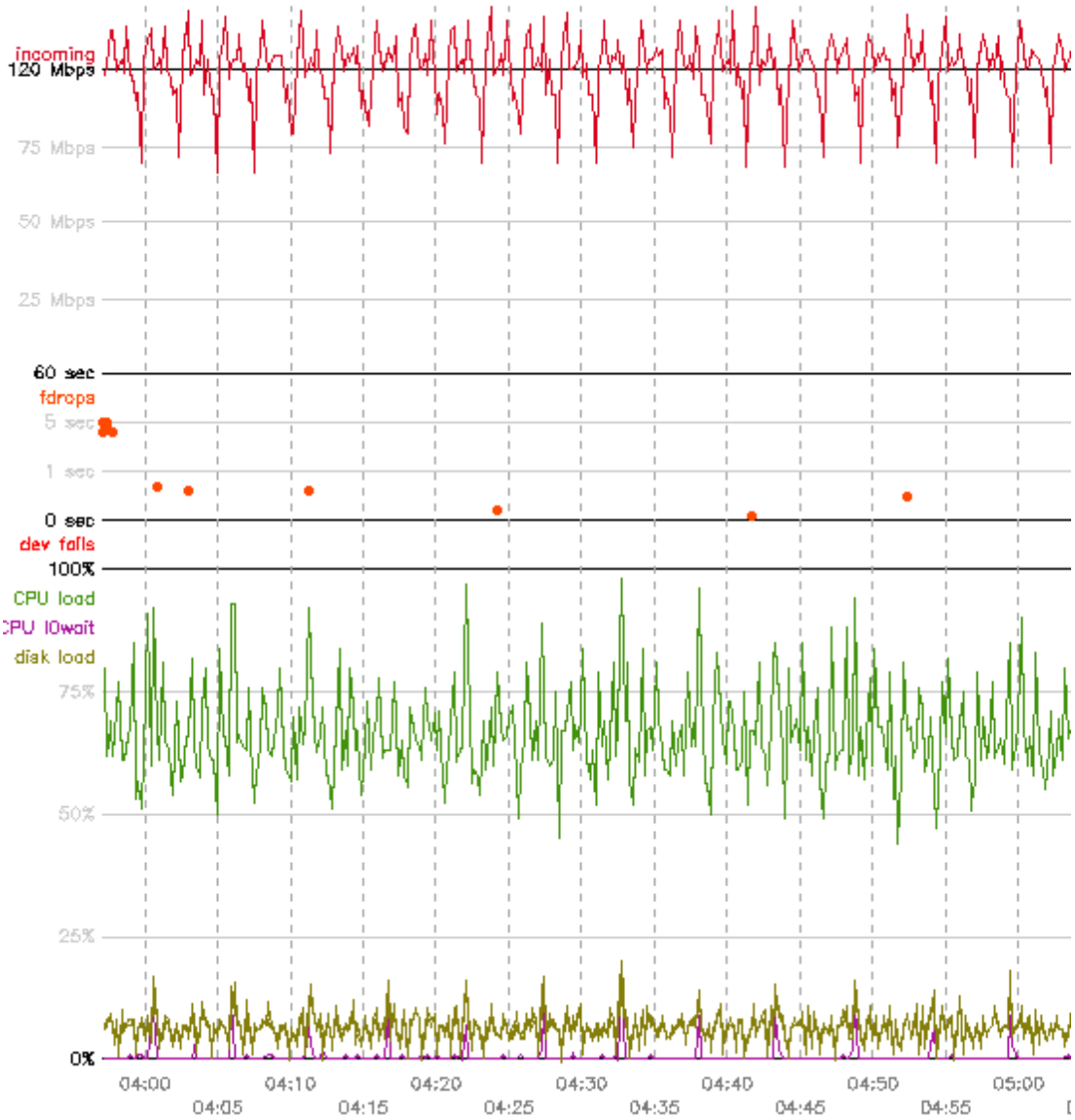


Figure 4.1: The final graph of a single experiment. This graph shows values of all the measured parameters in time. From the top: Incoming traffic (red), frame drop length (orange), CPU load (green), disc load (olive) and CPU I/O wait (purple)

4.4. Implemented Tasks and Pluggable Modules

All of the code was written by the Koukaam developers. They designed and implemented their benchmarks using the BEEN framework. Some of the work may seem as an unnecessary overhead but in the end it helped the programmers write better and more transparent code because the native BEEN API was often puzzling for them.

4.4.1. Python Libraries

In order to gain easier access to BEEN logging and *task* facilities, the Koukaam developers created a Python library ***BeenConnection***. This library provides better *task* property handling and easier access to the *Results Repository* than the standard `TASK` class that BEEN provides for Python code.

The IPCorder NVR can be controlled over the network by an XML API. BEEN *tasks* will send XML requests to the NVR and it will perform the requested action. In order to ease this communication, the developers created a Python library ***BoxLib*** for sending the most common XML requests. This second library reduced most of the benchmarking tasks to only a few lines of Python code.

4.4.2. Benchmarking Tasks

The first *task* that used the new BoxLib library was one that installed a specified firmware. The firmwares are generated by a server that distributes firmwares to all the IPCorders in the world. The ***koukaam-ipc-flash*** *task* takes the firmware name and the NVR address and credentials as its parameters. The *task* sends a series of XML requests to log in to the NVR and to tell it to download and install the given firmware.

The next *task* is ***koukaam-setup-fake-cameras*** which again sends a series of XML requests to the NVR to connect it to the streaming server. This *task* is parametrized by the NVR address and login credentials, by the number of cameras, their bit rate and their encoding. The NVR starts recording immediately after the cameras are set up. At this moment, we just have to wait for some time for the performance log to be filled with data.

The last benchmarking *task* that can sometimes be skipped is the one that simulates a client and measures the responsiveness of the NVR. The *task* ***koukaam-ipc-repeated-request*** is parametrized again by the NVR address and credentials, by the request that is to be sent, the number of repetitions and the waiting time between two requests.

4.4.3. Generator Pluggable Module

The BEEN *pluggable modules* need to be coded in Java. *Generator pluggable module* for this experiment does not create any *datasets*. Its configuration contains data for all the *tasks* mentioned above. The NVR address and credentials, the XML request data, firmware version and simulated camera data. This generator is able to perform one run of the experiment on a specified firmware version. In our first attempts, regression benchmarking is performed by running the same configuration on two different versions of the firmware and comparing the results. We did not

implement automatic firmware selection because there are too many versions and filtering them automatically would be more work than selecting versions by hand.

4.4.4. Evaluator Tasks

BEEN *evaluators* are *tasks* that read data from the *Results Repository*, perform the statistical calculation and then present the results in a readable form. In the previous use case we learned that using the *Results Repository* is very challenging and the results presentation still has to be programmed because BEEN does not provide any support for data presentation. We decided to bypass the *Results Repository*, store the results to the presentation server and use Django framework for easy creation of the presentation web. The schema of our database tables is displayed in tables 4.1 and 4.2. Notice that there is only one table field for all the results – raw and parsed data alike. This allows us to add more *evaluator tasks* later on without the need to change the database schema.

The raw results of our measurement are the performance logs that the NVR creates while it is recording. After the repeated-request *task* or simply-wait *task* finishes, the log is fetched from the NVR by the ***fetch-perflog*** *task*. This *task* uses the NVR's FTP interface and downloads the data to the web server where results are stored and processed.

The ***analyze-perflog*** *task* is started right after the fetch-perflog *task* finishes. It analyzes the log line by line, creates the statistics for CPU load, CPU I/O wait, disk load and incoming traffic. The *task* also reads all the frame drop reports and stores all the gathered data back in the MySQL database.

Table Column	Type	Description
name	`Varchar(60)`	Name of the NVR
platform	`Varchar(60)`	The NVR type
address	`Varchar(60)`	The IP address of the device
username	`Varchar(40)`	The NVR credentials
password	`Varchar(40)`	The NVR credentials
streamserver	`Varchar(60)`	The address of the streaming server for the NVR

Table 4.1: The structure of the database table ***box***. This table is designed for storing the information about the NVRs that are used for testing.

Table Column	Type	Description
created	`DateTime`	When the experiment was created
description	`Varchar(255)`	Short description of the experiment
box	`Foreign key`	Reference to the box table
firmware_version	`Varchar(120)`	Hash identification of the used version
tasks	`Varchar(255)`	Newline separated list of tasks to run
camera_count	`Integer`	Numbers of virtual cameras to test
camera_codec	`Varchar(10)`	The used video encoding
camera_bitrate	`Integer`	Bitrate per camera
request	`Text`	The XML request that will be sent to the NVR
count	`Integer`	How many times to send the request
waitTime	`Integer`	Time between requests
state	`Varchar(12)`	The current state of the experiment (scheduled, running, ...)
progress	`Varchar(255)`	NVR type
_results	`Blob`	Field where all the results are stored in a serialized form

Table 4.2: The structure of the database table ***BenchmarkRun***. This table is designed for storing the information about an individual experiment and its results.

4.5. BEEN Deployment

4.5.1. Operating System Configuration

As in the Seznam.cz case, we chose to dedicate a virtual server for the BEEN installation. The server ran standard Debian 6.0.3 distribution with Java 1.6, Tomcat 6, Python 2.6.6 and git version control system. With the experience from the previous experiment, the deployment was much less challenging. We ran into the same problems with domain name resolution and JVM configuration as in the previous deployment scenario.

4.6. Measured Data

4.6.1. Final Graph

For each measurement the benchmark produced a graph similar to figure 4.1. This graph is taken from a KRR-424 that recorded 24 virtual cameras each using h.264 codec and 4 Mbps bit rate. In the graph we can see quite regular course of the experiment. The incoming traffic oscillates near to 100 Mbps, bellow it we can see a few frame drops happened but the CPU load was always around 70% and disk load even lower at about 10%.

Guessing averages from this picture is not very accurate so the experiment overview page contains the exact statistics. We compute the average, median, minimal and maximal values for each of the measured properties. The result page also displays a histogram for each of the measured properties for better visualization of the results.

As we will see in the next chapter, only the numbers and histograms of each parameter do not give us the complete information. The result graph can show us error patterns occurring during the experiment which makes it invaluable.

4.6.2. Discovered Issues

The measurements helped Koukaam a.s. discover some serious issues with the firmware that could not be detected otherwise. In the figure 4.2. we can see a graph showing the course of another experiment. This graph displays several issues at once.

Figure 4.2: Final graph of an experiment that illustrates several issues discovered by the measurement.

The most notable problem is the long frame drop sequence. We discovered that the measured frame drops were not caused by actual gaps in the recorded video but rather by a mistake in the video frame timestamp[1] detection. As defined in chapter

[1] ***RTP timestamp*** is a form of relative clock that can be used only to determine the relative time elapsed between two frames in a RTP stream.

4.3.2. frame drops were reported when there was too much time between two frames. Strange things sometimes happen in the streaming servers used by cameras therefore the NVR has fault tolerance routines. These routines detect and deal with any frame timestamp that does not fit in the video frame sequence. They compute the range where a correct timestamp value will be next, wait for a value that can be used and ignore the invalid values provided by the streaming server. These algorithms were not prepared for an error situation when the streaming server unexpectedly reset the RTP timestamp and started from a random value. In this case we can see that the streaming server suddenly restarted the RTP timestamp sequence to a value about 38 minutes in the past.

Another potential problem that we can see in the graph is constantly high CPU load property. It ranges from 74% to 99% with average and mean values equal to 96%. This experiment was ran on the weakest hardware, the KNR-090. The device can support up to 4 cameras with 4 Mbps each. The Incoming trafic averaged at 15.9% so we tested the device at the edge of its capabilities and we can see that there was no room for increasing the device capabilities without serious firmware optimizations.

The last issue to that stands out in the graph is regularly increased disk load with values around 8% increasing to 50% every 5 minutes. Also a rather high CPU I/O wait property that rapidly increases every minute. The reason is that the device was tested with a full HDD. To allow continuous recording on the NVR there is a routine that runs approximately every 5 minutes that deletes old recordings in order to make a room for the incoming video. We can see that this routine is a clear candidate for optimization because it is probably the biggest reason behind the disk load increase. The CPU I/O wait is probably caused by a watchdog task which performs regular checks whether all vital processes are running.

4.7. User Experience

We will summarize this chapter in one block because in Koukaam a.s. the developers were also the users of BEEN. There were again only 2 developers from the company which led us to use the regular feedback model again. We have used BEEN for a much simpler benchmark which yielded surprisingly different user experience than in our other tryout scenario.

One similarity to the previous deployment scenario is that overall BEEN did not pass as a useful tool. Again the developers ran into insufficient documentation, missing *tasks* for performing simple common tasks like manipulation with a results file, and overcomplicated work with the *Results Repository*.

A unique perspective of independent developer, who was trying to get to know BEEN for the first time, revealed more problems that I was not able to see due to my experience with the framework. The first thing that the developers were puzzled with is that BEEN can do nothing on its own. There is only the SimpleTest benchmark which only illustrates how the *tasks* work and can be used to test BEEN's installation. There are no quick start options for real benchmarking with BEEN. Several custom *tasks* have to be implemented before we get useful data.

Another problem that we suspected was real but that we never acknowledged was the complicated API that BEEN provides. A simple API does not need an extensive documentation and is easy to understand and use. BEEN's complicated API

is not only hard to understand but it also restricts the programmer's options. The *task* author is often forced to do things just to satisfy the API. An example of this is the need to write the *task generator* in Java or the need to have every scripted *task* accompanied with XML meta data and packed into a BEEN package file. Another kind of unnecessary complication is revealed when programming the *generator pluggable module*. The developer is forced to set *task's* parameters which he would like to ignore but they have no defaults. Along with the fact that it is hard to find the right function in the overly extensive APIs it is best illustrated by an excerpt from the code in the table 4.3.

```
56: Condition beenCondition =
      new NotEqualCondition<String>("name",
      "this.really.stupid.system.has.no.true.value.so.we.have.to.wr
ite.essay.about.nonsense");
…
77: task = createTask("koukaam-setup-fake-cameras", beenCondition,
"cam-setup");
```

Table 4.3: An excerpt from file `RequestBenchmarkGenerator.java` in the Java package `com.koukaam.been.module.ipc.request.generator`. The developer needed an `AlwaysTrueCondition()` which is present in BEEN, only in a different package than the condition used.

A feature that the developers noticed is that both simple and difficult *tasks* have very similar complexity in BEEN. Difficult *tasks* for distributed measurements have a solution very similar to running a simple *task*. This is why we wanted to use BEEN on more complex measurements. Unfortunately the perceived difficulty causes that BEEN will not be used in a more complex scenario where it could be an effective tool. It gets dismissed in the tryout period because it is unusually difficult to perform simple tasks.

In the end the developers from Koukaam a.s. concluded that BEEN offers many things that are potentially useful but they are not suitable for their use cases. The most important functionality that BEEN provides as a benchmarking environment to them can be replaced by a job scheduler like the Linux cron daemon. The rest of the benchmark functionality has to be programmed from the start even when using BEEN.

On the conceptual level, performance evaluation was recognized as a new way of testing the software. The developers appreciated the data they gained from the measurements and the benefits that the results had for their problem solving. They continued in developing additional benchmarks in their own benchmarking tool that emerged from our experiment and they added performance measurements into their regular testing routines.

4.8. Replacement Framework: Kooň

After trying out the BEEN framework, the developers from Koukaam a.s. extended the code they had used in BEEN to create their own benchmarking framework. They identified the functionality that BEEN provided, used only what they needed and left out most of the things that added an unnecessary complexity to BEEN.

4.8.1. Characteristics

Kooň is a benchmarking framework developed in Koukaam a.s.. It is coded in Python, using Django South[1] for creating web interface and MySQL as a data storage. Biggles library is used to plot the output graphs. Biggles depends on libraries that are available only for Linux, which binds Kooň to only that one type of operating system.

Kooň emerged from the BEEN tryout. At first, we used BEEN facilities, created BEEN *task* packages and ran them using BEEN's *Benchmark Manager*. When the result analysis and presentation was implemented on a separate server, the Koukaam developers started to shift additional BEEN functionality there, extracting the functions that were useful for the company and abandoning everything that added unnecessary complexity. Eventually they programmed an environment that allows them to dynamically select and run Python tasks.

4.8.2. Similarities to BEEN

Since Kooň is based on the BEEN framework, it shares its philosophy and some of its features. The most notable similarity is that both frameworks use *tasks* as the basic execution unit. In Kooň the *tasks* are Python objects contained in tasks directory that all have a common ancestor and interface. *Tasks* have method `perform` which contains all the benchmarking code and a method `contribute` which returns data that are to be saved among other *tasks'* results.

Another less obvious but important similarity is a large amount of flexibility that both the frameworks provide. In BEEN the developers often had to overcome the extensive levels of abstraction that the framework uses to provide as much flexibility as possible. Kooň is created specifically for comparison benchmarking of a limited family of products. In the limits defined by the framework's purpose it allows for rather big flexibility as to how the measurement is performed and what the results data are. The measurement may be carried out by any Python *task* and the results data only need to be serializable values stored in a list. This simplification is possible only because we do not need to search the results and we have a fixed set of the experiment meta data that can be searched.

A feature that is configurable in BEEN but already fixed and tied to the experiment type in Kooň is the host management. All the evaluation and support *tasks* are ran only on the server where Kooň is deployed. There still may be multiple tested devices and Kooň makes sure that there is at most one experiment running on each device. This is possible only because we removed a lot of flexibility from the experiment.

Using BEEN terms, Kooň is a very configurable *task generator* that knows only one analysis.

[1] Django is a high-level Python web framework used for fast deployment of web applications. South is an intelligent schema and data migrations for Django projects. We used South to simplify future extensions of Kooň.

Chapter 5: Deployment Summary

5.1. Initial expectations

Before deploying BEEN in a an industrial environment, we expected that the companies will not be very enthusiastic about trying out the BEEN framework. We expected only limited support from their developers. This meant that most of the benchmark *tasks'* design and implementation would not be done by the company team but by me instead.

We planned to carry out a benchmark that would be either distributed in nature or it would be a regression benchmarking. It would also be a rather complex measurement to justify the use of such a complex benchmarking tool. On the other hand, the complexity of the benchmark would have to be limited by the time that the company would be willing to give as a testing period for BEEN.

At the end, we would collect the measurement outputs and analyze their value for the company. The company developers would be given a questionnaire that would ask about their experiences with the BEEN framework and how they viewed the usability of this benchmarking tool.

The output of this thesis should be a comprehensive evaluation of the BEEN framework with regards to the corporate needs. Also we should suggest improvements to BEEN that should be taken into consideration to make it more suitable for general use.

5.2. Deployment Aspects

We managed to try out the BEEN framework in two different companies. As expected, neither of the companies was enthusiastic about this complex toolkit that required a lot of learning before the developers were able to use it. Let us review the outcomes of these deployment scenarios.

5.2.1. Measured applications

Each company had different benchmarking needs. The tested server application in Seznam.cz aimed to utilize BEEN's ability to manage distributed measurements with multiple clients connected to a single server machine. Regression benchmarking was not the main goal but we needed a lot of computing power to put enough strain on the tested server. That was to be achieved by using multiple clients synchronized by BEEN.

At Koukaam a.s. the main focus was to compare the performance of different versions of the IPCorder firmware. We wanted to find performance problems and ways to improve the system's performance by tuning parts of the system and repeating the same measurement after each change.

Each of the experiments tested different aspect of the BEEN framework but neither of them was suitable as a complex tryout of this powerful tool. We realized that companies have generally two problems with this complexity. One problem is

that the companies need to quickly evaluate a new tool on a simple test case. Equally problematic is that most of the benchmarking experiments are simple in nature and the company needs one framework for all their benchmarking tasks, simple and complex ones alike.

5.2.2. Teams

In Seznam.cz the team was very busy and didn't even believe that the measurement was necessary. They let me do the job in their company and believed it would bring them added value in form of completed benchmarks with only a minimal effort. Developing the benchmarks was not their priority and therefore they assumed only a consultant role in benchmark development. This attitude eventually led to the early termination of the project.

On the other hand in Koukaam a.s. the developers took the initiative and learned how to write *tasks* for the BEEN framework themselves. They implemented all the necessary tasks using mostly the documentation. Therefore we got the chance to learn what problems would a real third party developer encounter. Their higher investment kept them at the task until they performed the first measurements.

In both scenarios we encountered very similar problems with BEEN deployment and tasks creation. Surprisingly neither of the teams was able to get the first measured data in less than 2 months where I expected the results within a few weeks. Also, neither of the teams considered BEEN a usable benchmarking tool.

5.2.3. BEEN Tasks

In Seznam.cz the implemented *tasks* were able to build, setup, run and stop the tested server. They also could run one or more client applications. The focus of the measurement was in generating variable load on the server from multiple clients and measuring the impact on the server application. This use case was exactly as we hoped for in our initial expectations. It lacked the regression testing which is easy to add simply by testing again on a new version of the server application.

In Koukaam a.s. we already had an existing system for automated builds and the measured devices were able to install any firmware. The *tasks* only needed to tell the IPCorder which firmware to install and then they needed to set the device in a predefined standard state. The focus of the measurement was in performing the same test over and over on different firmware builds. This is the part which was missing in Seznam.cz test scenario.

In both cases we realized that BEEN is overly focused on automated benchmarking but does not provide any tasks to support it. It is not possible to create and try out a sequence of *tasks* without the need to code, compile and deploy a *generator pluggable module*. Debugging a *generator pluggable module* is one of the harder tasks in BEEN because *pluggable modules* can not be unplugged from a running service. This and number of additional minor inconveniences make the *task* sequence tests too complicated.

BEEN is also missing a standard set of the most basic *tasks*. Among the missed *tasks* there was a *task* that would upload a log file into the *Results Repository*, Download it from there for another *task* to process, run a shell script, send a XML request to a server, etc. As a former coauthor of BEEN I know that we never expected these particular tasks to be missed. We planned to implement utility *tasks* that would

not have much use in either of the test cases. The list included a SVN checkout *task*, a build *task*, and a ping request *task*. We never implemented those as well because the team disbanded before we managed to implement the *tasks*.

5.2.4. Measurement Results

In Seznam.cz the project was stopped before we managed to fully develop the evaluation part of the application. Unfortunately the complexity of the measurement required multiple runs to give sufficient amount of data, which we were not able to collect due to the early end of the project. We obtained only the preliminary data for client calibration. From the final server tests we acquired only raw outputs and partially analyzed data of a few tests. More calibration tests on the server machine and many more experiments would be necessary to finish the tests as we intended.

In Koukaam a.s. we had a simpler scenario and even an individual run was sufficient to provide meaningful results. Seeing the results after the first successful run had a positive effect on the continuation of the project. We created a results page and continued in developing the benchmarks. In the end the results page turned into a new framework that was simpler to manage and extend than BEEN. This new framework provided the same functionality as the team previously used in BEEN and required much less from the developer.

The main drawback of BEEN framework is that it has no ability to interpret even the simplest results. Both the deployment projects required the BEEN users to implement the whole presentation layer for the measured results. In the second case it was even realized that the presentation layer contained so much logic that only a small amount of work can turn it into a benchmarking framework by itself.

5.3. Company Costs and Benefits

5.3.1. Company Costs

Seznam.cz invested over 50 man-hours of their programmers into assisting with this thesis. The BEEN task development took additional 150 man-hours. We used about 20 hours on the server machine where the tested server was run. Additional 40 man-hours were spent on evaluation of the preliminary outputs and refining the R scripts that generated the experiment output graphs. The estimate of the work remaining is additional 40 man-hours before the final graphs can be shown. Overall this benchmark's costs are 280 man-hours plus the measurement time on the server machine.

In Koukaam a.s. BEEN installation and deployment took only 5 hours of my time. The rest of the work was carried out by two very experienced developers. They had to learn BEEN API and create the benchmarking tasks. It took them over 270 man-hours to get the results as described in chapter 4.6.

5.3.2. Company Benefits

Seznam.cz canceled this project before we finished the last evaluator task and therefore we didn't have the desired output. Lack of the final numbers and graphs caused that we did not have the chance to fully examine the benefits of the

benchmark. The only benefit of this tryout to the company is that it realized that BEEN is not a good toolkit for them.

In Koukaam a.s. the developers learned to appreciate the importance of performance measurements and adopted benchmarking into their everyday work. BEEN taught the developers how to structure a scalable benchmark. Some of the key features from BEEN can be found in the new Kooň framework. Most notable is the usage of small tasks used together to perform the measurement and data analysis. Another aspect of the new framework inspired by BEEN is its flexibility. BEEN showed the team how to structure the benchmarking process and how to make their own framework extensible.

5.4. Retrospective Summary

There were two independent and very different cases of deploying the BEEN framework. Each deployment scenario allowed us to look at the BEEN framework from a specific side and together they covered all the aspects of the framework that we outlined in the initial expectations. We had the company developers only supervising the benchmark creation as well as fully engaged in the process. We tried a complex distributed measurement as well as repeated tests on multiple versions of the same product.

At the end we determined that the costs associated with the BEEN trial are too great for a company to tolerate the continuation of benchmark development using BEEN. The cost to the company is easily quantifiable and by no means it is negligible. The benefits are mostly educational and do not bring any competitive advantage that would justify the cost.

We also outlined reasons why continuing the development of BEEN as it is designed right now is not effective. When choosing the benchmarking experiments we realized that a lot of benchmarking in a company is simple in nature. Too much abstraction, focus on generalization and universality and almost no support for the simple applications in BEEN makes a proprietary benchmarking framework more eligible.

Chapter 6: Conclusions

We tried to use BEEN in two different scenarios. Each of the deployment scenarios focused on a different aspect of the framework. The original premise was that using BEEN for this task should be easier than writing a new benchmarking software from the scratch. In Seznam.cz we came to the conclusion that using BEEN is not a viable option. In Koukaam a.s. we proved that writing and maintaining a proprietary benchmarking framework is simpler than using BEEN.

Neither of the companies that tried BEEN kept it as their benchmarking platform. This leads us to a conclusion that we should reconsider the way how BEEN is developed. Right now it is not on the right track to be a universal benchmarking framework. With the experience gained by these deployment scenarios we should also reconsider whether there can even be a universal benchmarking tool.

6.1. BEEN Usability

Overall we can see that the main problem of BEEN is the lack of focus of the toolkit. The framework claims it is simple, extensible, multi-platform and powerful in general. In reality it has severe shortcomings in all the areas. More importantly these are only general qualities of any software. Neither of them solve any problem that a business may have.

BEEN started to be developed as a regression benchmarking environment defined in [11]. Our research indicates that the last BEEN development team, which I was a part of, got distracted by adding in more possibilities rather than limiting them to get a standardized environment. Our belief that widening options would lead to more universal tool have now been proven wrong. Placing restrictions on the benchmark and its results is necessary in order to be able to implement a comprehensive set of reusable tasks.

We will conclude this thesis by proposing improvements to the existing BEEN framework that will improve its usability in industrial environment similar to where BEEN was tested.

6.1.1. Clarify the Purpose of BEEN

Currently it is not clear what problems BEEN tries to solve for the user. It simply states it is a benchmarking environment but it does not say what it can do for the person who decides to use it. When the purpose is stated, it is easier to focus the development to fulfill it. BEEN lacks this focus and therefore it appears to only have many imperfections in many areas.

On the web page of the BEEN project we can find that "*BEEN is a generic tool for automated benchmarking in a heterogeneous distributed environment. […] BEEN has been designed to facilitate automated detection of performance changes during software development (regression benchmarking).*" [12] This statement expresses that this tool is extensible and has some positive qualities of a good software. It just needs to state what kind of environment BEEN is good for, what kinds of applications it is able to measure efficiently, what kinds of results it can provide, etc. The more parameters are mentioned, the better. Stating that the

framework can for example "*measure responsiveness of any server with a generic XML interface in a heterogenous environment"* would make it much more focused.

6.1.2. Easy Installation

Both the companies used virtual servers for the main BEEN installation. Due to the complex nature of the framework it would be much easier for the users to provide a preinstalled virtual server that would contain a configured instance of BEEN ready to run. This will eliminate problems with system configuration described in chapter 3.7.

6.1.3. Bash and Python Tasks Support

Both the companies found it easier to write *tasks* in scripting languages due to their simplicity. The need to pack these scripts that contain only a few lines of code into been package with 2 meta data files and uploading them to a server is too much of an overhead. Most of the time it is easier to update the script than to push the updated version into BEEN.

The overhead associated with converting a script into a BEEN *task* leads the developer to creation of an universal *task* that will simply run a script from the host's file system. The script can then be easily updated but there is a high risk of changing the whole benchmark by a mistake.

BEEN would benefit from the option to upload scripts directly to the BEEN environment as *tasks*. The meta data required by BEEN may be contained directly in the script file as a comment.

6.1.4. Standard Results Format

Having a universal *Results Repository* for storing configurable result sets in various formats proved to be very difficult to understand. Most developers know SQL and basic variable types used in relation databases. The need to learn a new database system discourages them from using BEEN.

BEEN should do one of two things:

1) It should either release its control of its *Results Repository* by giving the benchmark authors SQL-compatible database to use as a *Results Repository*. This will allow easy data storage and retrieval by the user. Writing the evaluator tasks will not be a difficulty then.
2) Or it should define more restrictive *Results Repository* which will allow to write universal evaluator tasks. These tasks will provide some of the missing data presentation functionality.

6.1.5. Utility Tasks

The most missed feature in both the companies was a set of utility *tasks*. Configurable *tasks* that would checkout a revision from a SVN or GIT repository into a directory accessible by another *task*, *tasks* that would save files into the *Results Repository* and load them back to the local disk. Evaluation *tasks* and plotting facilities that would ease the writing of *evaluators*.

Koukaam a.s. developers were puzzled when they learned that BEEN does not provide a place for the final presentation of results. Another server was required to serve the graphs and analyzed data.

6.1.6. Stability

While debugging the *tasks* we had to restart the core BEEN *services* several times a day in both companies. Debugging a *generator pluggable module* requires restart of the *Benchmark Manager* every time when a new version is uploaded into the *Software Repository* because the *pluggable module* can not be unplugged.

6.1.7. Task Running and Debugging

When debugging a *task*, it is often required to run it several times by hand before it can be incorporated into a *generator* to start it automatically. *Task Manager* does not allow to run a *task* more than once in one context which complicates the test runs. This is caused by the task tree feature which proved to bring more problems than benefits. The task tree keeps information even about deleted *tasks but* an attempt to access them causes error.

Besides being able to run a *task* repeatedly, it is useful to be able to debug a *task*. Debugging a Java *task* is surprisingly easy. Unfortunately both the companies worked with *tasks* scripted in Bash and Python. The only debugging tool for these *tasks* are the *task* logs. The logs are buffered, therefore sometimes when the *task* crashes, the log is missing the last few entries and sometimes is even empty. Debugging *triggers* in the *Results Repository* is near to impossible no matter what programming language the author uses.

6.1.8. Constant Development

Easily overlooked but important drawback is that BEEN misses a maintenance team. All software is unusable without continuous development and support. If there was a team assigned to development of BEEN framework then Seznam.cz developers would consider using it for their benchmarks. Without a stable support it is not a viable option for them.

Chapter 7: Bibliography

[1] Benchmark (computing). In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, September 2012. Available from: http://en.wikipedia.org/w/index.php?title=Benchmark_(computing)&oldid=515269681

[2] Software regression. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, November 2012. Available from: http://en.wikipedia.org/w/index.php?title=Software_regression&oldid=523906743

[3] Kalibera Tomáš, Tůma Petr. Precise Regression Benchmarking with RandomEffects: Improving Mono Benchmark Results. In *proceedings of Third European Performance Engineering Workshop (EPEW 2006)*. Budapest (Hungary): Springer-Verlag, Berlin. LNCS 4054, ISBN 3-540-35362-3, ISSN 0302-9743, pages 63-77. Available from: http://d3s.mff.cuni.cz/publications/download/KaliberaTuma-PreciseRegressionBenchmarking.pdf

[4] Software framework. In: *Wikipedia, The Free Encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, November 2012. Available from: http://en.wikipedia.org/w/index.php?title=Software_framework&oldid=524232123

[5] The Use of Questionnaire Methods for Usability Assessment. In: *Software Usability Measurement Inventory* [online]. Cork (Ireland): Dr Jurek Kirakowski, 1994. Available from: http://sumi.ucc.ie/

[6] Motivation for Middleware. In: *What is Middleware* [online]. Paris (France): OW2 Consortium, 2007. Available from: http://middleware.objectweb.org/

[7] Seznam.cz . In: *Wikipedia, The Free Encyclopedia.* [online]. San Francisco (CA): Wikimedia Foundation, June 2010. Available from: http://en.wikipedia.org/w/index.php?title=Seznam.cz&oldid=370317809

[8] Poisson process. In: *Wikipedia, The Free Encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, November 2012. Available from: http://en.wikipedia.org/w/index.php?title=Poisson_process&oldid=522081217

[9] Fork bomb. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2011. Available from: http://en.wikipedia.org/w/index.php?title=Fork_bomb&oldid=442967132

[10] Application programming interface . In: *Wikipedia, The Free Encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2012. Available from: http://en.wikipedia.org/w/index.php?title=Application_programming_interface&oldid=526494256

[11] Kalibera, Tomáš. Regression benchmarking environment. In *WDS'04*. Prague: MatfyzPress, Charles University. pages 174-178. Available from: http://d3s.mff.cuni.cz/publications/download/Kalibera-WDS04.pdf

[12] About BEEN. In: *BEEN - Benchmarking Environment* [online]. Prague: Department of Distributed and Dependable Systems, 2010. Available from: http://been.ow2.org/